Jtalis Manual

Jtalis Manual, Vesko Grorgiev, FZI, Karlsruhe

Contents

Change Log

Author	Date
Vesko Georgiev	22-Jun-2011
Vesko Georgiev	2-Aug-2011
Vesko Georgiev	26-Aug-2011
Vesko Georgiev	11-Dec-2011

About

Jtalis is Java wrapper for ETALIS engine. It provides convenient way for getting events in and out of ETALIS Prolog core. This project uses Maven 2 for build, so please follow the instructions below.

How to setup and build

First you have to check out the sources from http://etalis.googlecode.com/svn/jtalis. It is very important that your SVN client supports external fetching because the needed ETALIS Prolog sources are being externally fetched into *jtalis/jtalis-core/src/main/resources/com/jtalis/core/config/etalis/src/*. Without ETALIS sources Jtalis will not work. If your SVN client does not support external fetching, you can just copy ETALIS sources in the previously specified directory.

Because Jtalis binds Java with Prolog environment it requires 3rd party libraries to do so. Two commonly used are Interprolg and JPL. JPL is standardly distributed with SWI Prolog, it is faster than Interprolog, therefore it is used by default in Jtalis.

Windows XP/7

First you need to set SWI_HOME_DIR environment variable to point your SWI Prolog installation directory. Afterwards, double click to start *scripts/build.bat.* The script will configure the environment, install the necessary dependencies in Maven repository and build Jtalis sources.

NB! On 64bit Windows, 64bit SWI Prolog has known issues with JPL. Therefore it is mandatory to work with 32bit bit version SWI Prolog, and respectively with 32bit Java.

Mac OS X / Linux

Start *scripts/build.sh*. You might be requested for the SWI Prolog installation directory. The script will configure the environment, install the necessary dependencies in Maven repository and build Jtalis sources. Note that the script will add SWI_HOME_DIR variable in your ~/.profile, because it is usually needed by JPL.

Using Eclipse

If you are using Eclipse with m2eclipse maven plugin you must specify JNI library path for the native JPL library. If you are running Mac OS X the name of the variable should be DYLD_LIBRARY_PATH, for Linux - LD_LIBRARY_PATH. You can do that as shown in *Fig. 1.*

To run the examples or any other Java class that uses Jtalis, you need to put in VM arguments the JNI library path, e.g. on Windows it could look something like this: *-Djava.library.path="C:/Program Files/pl/bin;C:/Program Files/pl/lib"*

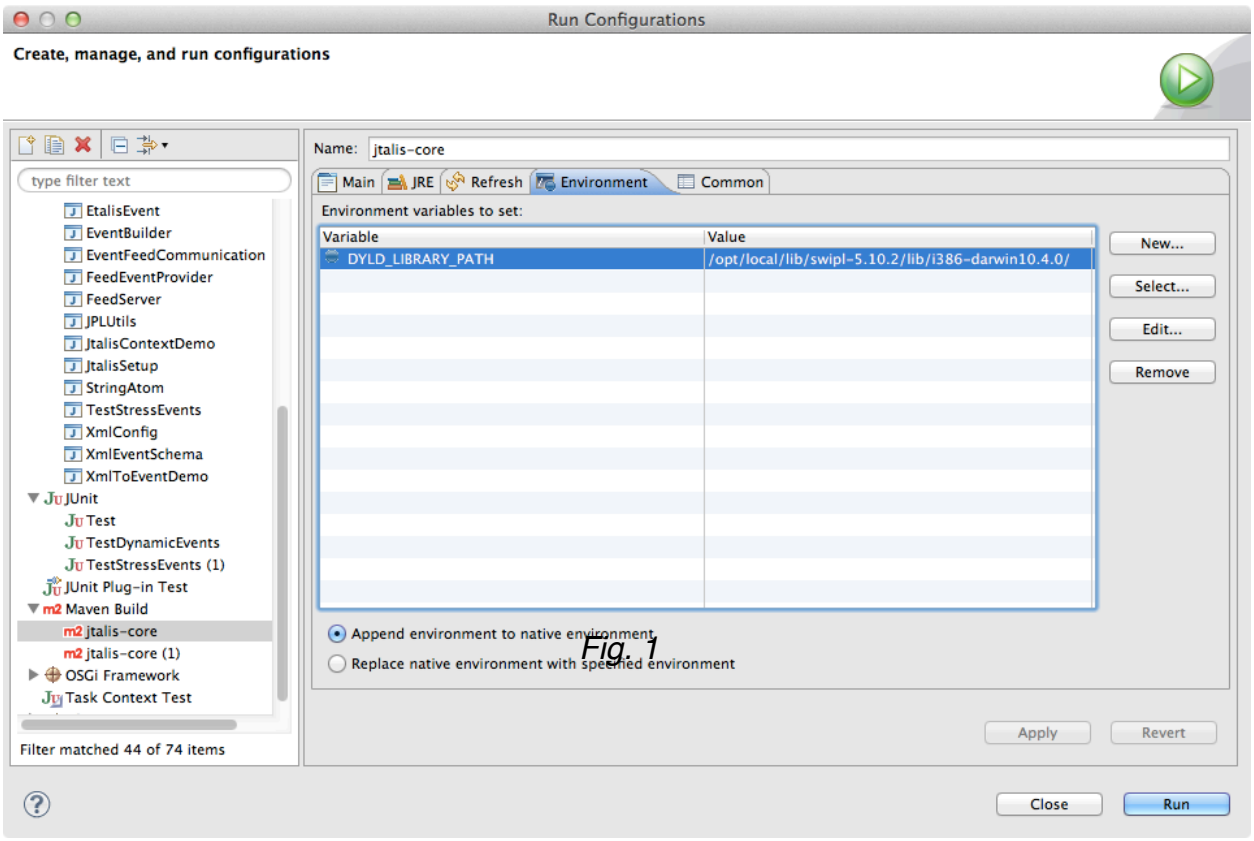

Fig. 1

Event Distribution

The events are provided by Input Event Providers, which are submitting events simultaneously in one single Input Event Queue. Depending on the Output Event Trigger (Event name pattern - could be either for complex or simple events) events are being fired back from ETALIS to the EtalisEventListener, which is responsible for dispatching the events to all Output Event Queues. This means each event is dispatched to all queues. But all Output Events Providers which are registered on one a single queue will be competing for the events, therefore only one provider will process a given event. For more information on how to control the event distribution programmatically, you can read the Java Doc for `com.jtalis.core.JtalisContext`.

Figure 2 shows the events distribution.

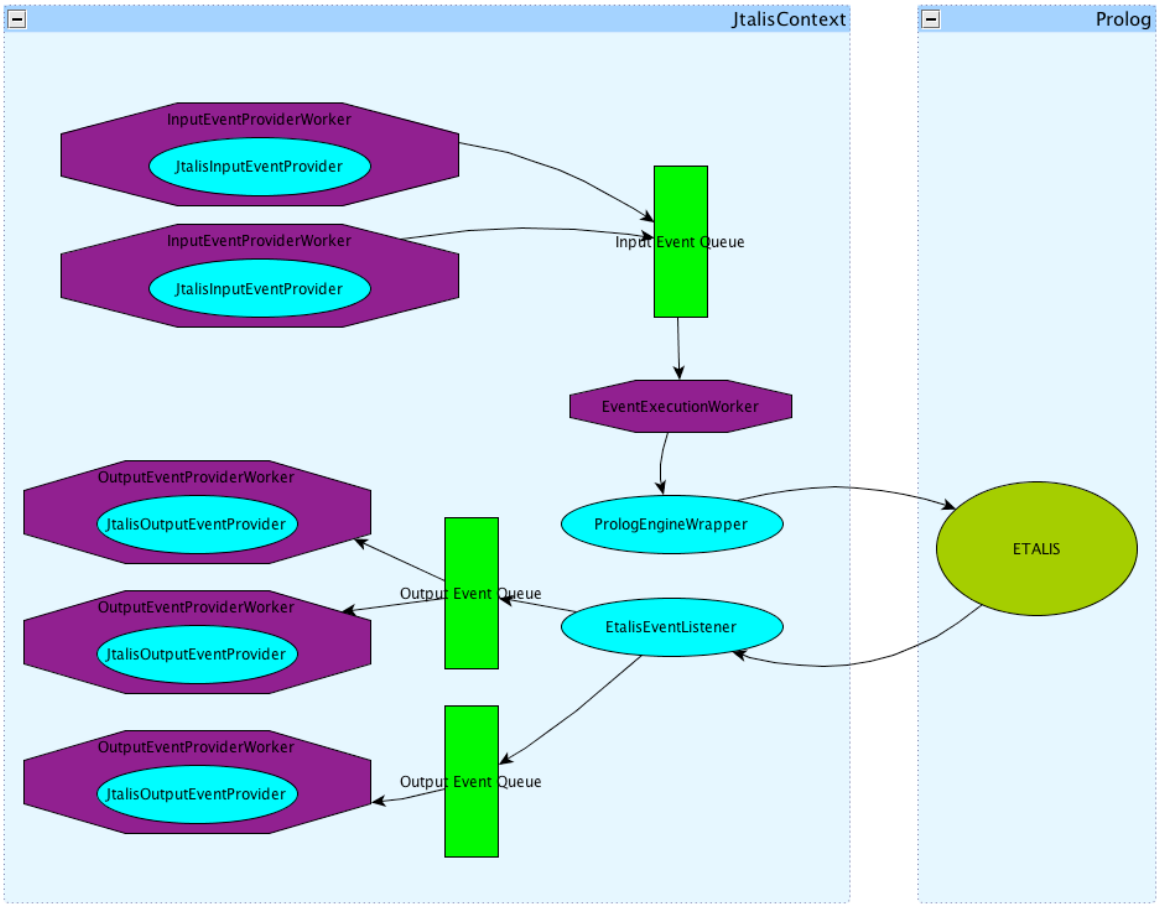

Fig. 2

Running the engine using XML configuration

You can run a Jtalis configuration without writing any Java code. Instead, you can specify all the parameters within an XML file and user *scripts/jtalis-run.sh/bat* after. The XML Schema for the configuration is located here: XmlConfig.xsd. The following parameters could be specified:

- **<debug>** - boolean, specifies whether the engine wrapper should log each prolog predicate that is being executed

- **<consult>** - list of additional prolog files to be consulted. e.g:

```
<consult>
        <file>additional.p</file>
        <file>other.p</file>
</consult>
```

- **<flags>** - list of ETALIS flags to be set. e.g:

```
<flags>
	<flag>
		<name>param</name>
		<value>value</value>
	</flag>
	<flag>
		<name>other</name>
		<value>someValue</value>
	</flag>
</flags>
```

- **<predicates>** - list of predicates to be executed beforehand. e.g.

```
<predicates>
	<predicate>assert(parameter(1))</predicate>
</predicates>
```

- **<compileEventFiles>** - list of event files to be compiled: e.g.

```
<compileEventFiles>
	<file>sequence.event</file>
	<file>other.event</file>
</compileEventFiles>
```

- **<eventTriggers>** - list of event patterns to be triggered for the output events e.g.

```
<eventTriggers>
	<trigger>a/_</trigger>
	<trigger>b/_</trigger>
</eventTriggers>
```

- **<eventRules>** - list of event rules to be loaded e.g.

```
<eventRules>
	<eventRule>
		<consequence>c(X)</consequence>
		<antecedent>a(X) seq b(X)</antecedent>
	</eventRule>
</eventRules>
```

This will generate c(X) <- a(X) seq b(X)

- **<providers>** - list of providers. Each provider has Java **class** name and **type** - *input*, *output* or *both*. If type is omitted then the concrete type of the provider is taken into account - if it implements JtalisInputEventProvider then it is added as input provider, for output type is respective. There is also **regex** which is taken into account only if the provider is registered as *output* or *both.* It specifies regular expression for the event name and controls which kind of events does this provider accept. You can also

specify parameters for the provider. They are actually all fields annotated with com.jtalis.core.config.annotations.ConfigParam and will be bound to the XML values by Jtalis. e.g.

```
<providers>
	<provider type="input" class="com.jtalis.core.event.provider.CSVInputProvider">
		<parameters>
			<parameter name="url">localhost:8888</parameter>
		</parameters>
	</provider>
	<provider type="output"
class="com.jtalis.core.event.provider.CSVOutputProvider" />
</providers>
```

- **<outputQueues>** - for each output provider from the **<providers>** section a separate Output Queue is created. However, if you want to create a single output queue for several providers you need to user this tag. You can have as many queues as you want, with many providers for each. Here the **regex** property is specified for the whole queue, and if given in the provider tag it is neglected.

```
<outputQueues>
	<queue regex="a">
		<providers>
			<provider class="com.me.MyProvider" />
			<provider class="com.me.MyOtherProvider" />
		</providers>
	</queue>
</outputQueues>
```

In the whole configuration if a relative path is given it is taken as a relative to the configuration XML file - **consult** and **compileEventFiles** tags, as well as any provider parameters bound to java.io.File fields.

Possible problems

1. `Exception in thread "main" java.lang.UnsatisfiedLinkError: no jpl in java.library.path`

Cause
Prolog libraries, which are placed in bin and lib directory path in your SWI installation directory (e.g. *C:\Program Files\pl\bin* and *C:\Program Files\pl\lib*) are not in your java.library.path.

Solution
May be the easiest way to fix this one is to put prolog bin and lib directory paths into your PATH environment variable. Another solution is to add:

```
-Djava.library.path="C:\Program Files\pl\lib;C:\Program Files\pl
\bin"
```

as a Java VM argument

2. `[FATAL ERROR: Could not find system resources]`

Cause
This is a strange error due to some queerness of SWI Prolog on some Windows machines. You can read the original FAQ about this one: http://www.swi-prolog.org/FAQ/FindResources.html.

Solution
In case of the first cause you can just add *SWI_HOME_DIR* in your environment.

3. `Can't load IA 32-bit .dll on a AMD 64-bit platform" error (Windows 64bit OS).`

Cause
The problem is pretty self-explanatory. Your java is 64-bit and you are trying to load 32-bit dll.

Solution
Use 32-bit Java with 32-bit Prolog, or 64-bit Java with 64-bit Prolog. In case you change your Prolog to 32/64 bit, don't forget to change your environment variables to point to the new one.

4. Other problems may be caused by running Jtalis with user that does not have sufficient rights over Java and Prolog installation directories. Other could be due to mixing 64bit Java with 32bit Prolog or vice versa.